Twisting Additivity in Program Obfuscation

Mila Dalla Preda[1], Wu Feng[2]*, Roberto Giacobazzi[2], Richard Greechie[2], and Arun Lakhotia[4]

[1] University of Bologna, Bologna, Italy. dallapre@cs.unibo.it
[2] Louisiana Tech University, Ruston, LA, USA. {greechie,wfe002}@latech.edu
[3] University of Verona, Verona, Italy. roberto.giacobazzi@univr.it
[4] University of Louisiana, Lafayette, LA, USA. arun@louisiana.edu

Abstract. Additivity plays a key role in program analysis. It is the basis for designing Galois connection based abstract interpretations, it makes a Data-Flow Analysis (DFA) problem easy being convertible into a Kildall's general form, and provides a lattice-theoretic model for disjunctive analysis. In this paper we consider reversible transformers respectively making any monotone function additive and maximally non-additive. We show that, under non restrictive hypothesis, these transformers exist and that they provide a theoretical foundation for the obfuscation of DFA.
Keywords: Program analysis, distributive DFA framework, code obfuscation.

1 Introduction

Additive functions play a key role in programming languages and systems. They provide a model for transfer functions in what is known as *distributive framework* in Kildall's Data-Flow Analysis (DFA) [22], they provide order-theoretic models for predicate transformers in program and system verification methods such as *a la Hoare* verification logic, model checking and trajectory evaluation, and they constitute one of the key ingredients in Galois-connection based abstract interpretation [6]. In this latter case, additivity provides both the essence in having a Galois connection [9] and in case of both adjoint functions are additive, the essence of having a disjunctive analysis [17].

Functions can be modified by other functions, later called *transformers*. Modifying functions in order to obtain their additive counterparts, and conversely modifying them in order to make them maximally non-additive, provides an in-depth understanding of the role of additivity in programming languages and systems, unveiling how much of our understanding of programs and their semantics depends on additivity. For this reason we consider two basic transformers that return respectively the *residuated approximation* and the *residuated ceiling* of a given function f. The residuated approximation of a function f is the additive function that is closest to f [1], while the residuated ceiling of f is the largest function with the same residuated approximation of f. Thus, the residuated approximation of f represents the least modification of f in order to induce additivity, while the residuated ceiling of f represents the largest modification of f with the same residuated approximation. We prove that under non restrictive hypothesis these two transformers exist and can be computed easily for arbitrary functions on

* Funded by AFOSR grant FA9550-09-1-0715

completely distributive lattices. This generalizes a previous result on closure operators [16, 17] to arbitrary functions.

We apply these transformations to static program analysis by showing that these transformers provide simple methods for understanding some known code obfuscation strategies defeating DFA in the distributive framework. Obfuscating a program with respect to an analysis corresponds to deform the program in such a way that a maximum loss of precision is induced in the analysis [14]. When considering the obfuscation of a distributive DFA, this corresponds precisely to deform the transfer function of a program P in order to make it maximally non-additive. We show that this can be modeled precisely as the residuated ceiling of the semantics of P. The proposed application of the residuated ceiling to code obfuscation is language independent, and relies upon pure lattice-theoretic arguments on program semantics and analysis.

The paper is structured as follows: Section 2 recalls some background notions from lattice theory. In Section 3 we summarize the previous results on the transformers that modify functions in order to gain or loose additivity. We study the existence if these transformers by generalizing a previous result on closures. Section 4 shows how these transformers can implement an obfuscating algorithm defeating DFA for imperative programs.

2 Background

Mathematical notation. The notation $(C, \leq)$ denotes a *poset* C with ordering relation $\leq_C$, or when no confusion entails, simply $\leq$. The *down-set* of x in C is defined to be $\downarrow x := \{y \in C \mid y \leq x\}$. For $S \subseteq C$, we define $\downarrow S = \bigcup_{s \in S} \downarrow s$. A function $f : C \to C$ is *monotone* if $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in C$. For a function $f : C \to A$ between two posets, the function $f^{-1} : A \to C$ is defined as $f^{-1}(\downarrow a) := \{c \in C \mid f(c) \leq a\}$ for $a \in A$. For two functions $f, g : C \to A$, $f \sqsubseteq g$ denotes the pointwise ordering between f and g, i.e., $f(x) \leq_A g(x)$ for all $x \in C$, in this case we say f is *under* g. Given two functions $f : C \to B$ and $g : B \to A$ between posets A, B and C, we use $g \circ f : C \to A$ to denote the *composition of functions* f and g, i.e., $g \circ f(x) := g(f(x))$ for $x \in C$. Let $f : C \to A$ and $g : A \to C$ be monotone functions between two posets C and A; thus the pair functions (f, g) is a *monotone Galois connection* (alias a residuated-residual pair), for short a *Galois connection*, iff, for all $c \in C$ and all $a \in A$, $f(c) \leq_A a \Leftrightarrow c \leq_C g(a)$, or equivalently $c \leq g \circ f(c)$ for all $c \in C$, $f \circ g(a) \leq a$ for all $a \in A$, and for any $a \in A$, there exists $c \in C$ such that $f^{-1}(\downarrow a) = \downarrow c$.

A *complete lattice* is denoted by $(C, \leq, \vee, \wedge, \top, \bot)$ with ordering relation $\leq$, least upper bound $\vee$, greatest lower bound $\wedge$, top element $\top$ and bottom element $\bot$. A complete lattice C is *completely distributive* if it satisfies, for all index sets I and for all sets of selection functions sets F where $f(i)$ is some element in non-empty set $S(i)$ for each $i \in I$, $\bigvee_{i \in I} \bigwedge_{s \in S(i)} x_{i,s} = \bigwedge_{f(i) \in F} \bigvee_{i \in I} x_{i,f(i)}$. For example, the power set lattice $(\wp(X), \subseteq)$ for any set X is a completely distributive lattice [12]. It is *infinitely (join) distributive* if it satisfies that $x \wedge (\bigvee Y) = \bigvee_{y \in Y} (x \wedge y)$ for all $x \in C$ and $Y \subseteq C$. $x \ll y$ if for any chain $D \subseteq C$: $y \leq \bigvee D$ implies there exists $d \in D$ such that $x \leq d$. An element $x \in C$ is *compact* if $x \ll x$. The set of compact elements in a complete lattice C is denoted $K(C)$. C is *algebraic* if it is

complete and for any $x \in C$: $x = \bigvee(\downarrow x \cap K(C))$ [18]. Dual-algebraic lattices are defined by duality. In algebraic lattices, every element can be generated from compact elements by disjunction. A function $f : C \to A$ between two complete lattices is *additive* (resp. *co-additive*) if, for any $X \subseteq C$, $f(\bigvee_C X) = \bigvee_A f(X)$ (resp. $f(\bigwedge_C X) = \bigwedge_A f(X)$).

3 Inducing and removing additivity

In this section we follow Andrèka et al. [1] introducing the notion of residuated approximation of a generic monotone function on complete lattices and consider the case when residuated approximation is *join-uniform* [16], guaranteeing the existence of residuated ceilings.

Residuated approximation: making functions additive. It is easy to see that there exists the largest additive function ρ_f under a monotone mapping $f : C \to A$ where C and A are complete lattice, formally $\rho_f := \bigvee\{g : L \to Q \mid g \text{ is additive and } g \leq f\}$. Following Andrèka et al. [1] we call the function ρ_f the *residuated approximation* of f. They introduce a function $\sigma_f : C \to A$, called the *shadow* of f, that for any $c \in C$ is defined as follows:

$$\sigma_f(c) := \bigwedge\{a \in A \mid c \leq \bigvee f^{-1}(\downarrow a)\}$$

In [1] the authors prove that $\rho_f \leq \sigma_f \leq f$ and that σ_f is monotone. Su et al. [13] define the *umbral mappings* $\sigma_f^{(\alpha)}$ of f: for any ordinal number α,

$$\sigma_f^{(\alpha)} := \begin{cases} f, & \alpha = 0, \\ \sigma_{\sigma_f^{(\alpha-1)}}, & \text{for a successor ordinal } \alpha, \\ \bigwedge\limits_{\beta<\alpha} \sigma_f^{(\beta)}, & \text{for a limit ordinal.} \end{cases}$$

They also proved that: (1) $f \geq \sigma_f \geq \sigma_f^{(2)} \geq \ldots \geq \rho_f$, (2) there exists a least ordinal α such that $\sigma_f^{(\alpha)} = \rho_f$, and (3) as the ordinal number α becomes larger, the decreasing sequence $\sigma_f^{(\alpha)}$ converges to ρ_f. The least ordinal number α in point (2) is called the *umbral number*, u_f, of f. Point (3) shows that the residuated approximation ρ_f of f can be calculated by iteration of the shadow of f and the least iteration number is u_f. Of course u_f might be larger than 1. The efficiency of the iterative computation of umbral mapping is measured by the iterative number u_f. Andréka et al. prove a theorem in [1] which implies the following proposition.

Proposition 1. *Let $f : C \to A$ be a monotone mapping between two complete lattices. If A is completely distributive, then $u_f = 1$, i.e., $\sigma_f = \rho_f$.*

Residuated ceilings: removing additivity. Let us consider the problem of removing additivity by considering the largest function having the same residuated approximation of a given (additive) function. Following [16], a function $f : C \to C$ on a complete lattice C is *join-uniform* if for any nonempty subset $S \subseteq C$, if all the elements of S are

mapped by f to some c, then $\bigvee S$ is mapped by f to c. We note that a monotone function f is join-uniform if and only if $f(\bigvee f^{-1}(\downarrow f(x))) = f(x)$ for all $x \in C$, i.e., there is a largest element of C mapping to $f(x)$ for all $x \in C$. The residuated approximation of this function collapses to the given (additive) function if and only if the residuated approximator is join-uniform. Let C be a complete lattice. An element x in C is *completely join-irreducible* if $x = \bigvee S$ for $S \subseteq C$ implies that $x \in S$. Let $JI(C)$ be the set of completely join-irreducible elements in C, i.e. $JI(C) := \{x \in C \mid \forall S \subseteq C, x = \bigvee S \Rightarrow x \in S\}$. An element x in C is *completely join-reducible* if there is $S \subseteq C$ such that $x = \bigvee S$ and $x \notin S$. It has been proved that when C is a dual-algebraic complete lattice. Then, for all $x \in C$, it holds that $x = \bigvee((\downarrow x) \cap JI(C))$ [3]. We need the following two results in order to characterize the residuated approximation of a function in terms of the elements join-irreducible.

Lemma 1. *Let C be an infinitely distributive complete lattice and let $S \subseteq C$. If $x \leq \bigvee S$ and $x \in JI(C)$, then $x \in \downarrow S$. Hence $(\downarrow S) \cap JI(C) = (\downarrow \bigvee S) \cap JI(C)$ holds.*

Lemma 2. *Let $f : C \to A$ be a monotone mapping from an infinitely distributive complete lattice C to a complete lattice A. Then, for all $x \in JI(C)$, $f(x) = \sigma_f(x) = \rho_f(x)$.*

The following Theorem provides a setting in which there is a much simpler formula for the shadow. Moreover, in this setting the shadow of a monotone function f is, in fact, the residuated approximation of f. Thus, in this setting, we have a quite simple formula for calculating the residuated approximation of f. Recall that, if f is a function with domain C and $S \subseteq C$, then $f(S) := \{f(s) \mid s \in S\}$.

Theorem 1. *Let $f : C \to A$ be a monotone mapping from a dual-algebraic infinitely distributive complete lattice C to a complete lattice A. Then, for all $x \in C$,*

$$\rho_f(x) = \sigma_f(x) = \bigvee f((\downarrow x) \cap JI(C))$$

Let $\mathcal{M}_{C,A}$ denote the complete lattice of all monotone functions between the complete lattices C and A, where functions are ordered by the usual pointwise ordering. The *residuated approximator* $\rho : \mathcal{M}_{C,A} \to \mathcal{M}_{C,A}$ is a transformer that computes the residuated approximation of a given monotone function f, i.e., $\rho(f) := \rho_f$. Given a function $f \in \mathcal{M}_{C,A}$, we define the *residuated ceiling of f* as the largest function that has the same residuated approximation of f:

$$\omega_f := \bigvee \{\, g \in \mathcal{M}_{C,A} \mid \rho(g) = \rho(f)\}$$

This gives rise to another transformer $\omega : \mathcal{M}_{C,A} \to \mathcal{M}_{C,A}$ called the *residuated ceiling* defined by $\omega(f) = \omega_f$, for each $f \in \mathcal{M}_{C,A}$. Observe that the residuated approximator ρ is join-uniform if and only if for every monotone function $f : C \to A$ we have that $\rho(\bigvee\{g \in \mathcal{M}_{C,A} \mid \rho_g = \rho_f\}) = \rho_f$, i.e., $\rho(\omega_f) = \rho(f)$.

Theorem 2. *Let C be a dual-algebraic complete lattice and A be a complete lattice. If C is infinitely distributive, then ρ is join-uniform.*

In the setting of the last Theorem, we see that $\mathcal{M}_{C,A}$ can be partitioned into intervals $[\rho_f, \omega_f]$ in such a way that the transformers ρ and ω exchange the bounds of each of the intervals $[\rho_f, \omega_f]$ in the sense that $\rho(\omega_f) = \rho_f$ and $\omega(\rho_f) = \omega(f)$. In this sense, they are reversible transformers. The following example demonstrates that ω need not be join-uniform if it were to be defined on A^C (all functions from C to A). This is the reason we restrict the definition of ω to $\mathcal{M}_{C,A}$.

Example 1. Let C be the complete boolean lattice isomorphic to 2^3 with atoms a, b and c. Define $f_c : C \to C$ by $f_c(b \vee c) = a$, $f_c(a \vee c) = b$, $f_c(a \vee b) = 1$ and $f_c(x) = x$ for $x \in \{0, a, b, c, 1\}$. Since $f_c(c) = c \not\leq a = f_c(b \vee c)$, the function f_c is not monotone. Using the symmetry of C we may define f_a and f_b similarly. One easily calculates that, for each $f \in \{f_a, f_b, f_c\}$, $\rho_f = \sigma_f = 0_C$ and if $h := \bigvee\{g : C \to C \mid \rho_g = \rho_f\}$, then $h(x) = 1$ for $x \in \{a \vee b, b \vee c, c \vee a\}$, otherwise $h(x) = x$. It follows that $\rho_h(x) = x$ for $x \in C$. Thus $\rho(\bigvee\{g : C \to C \mid \rho_g = \rho_f\}) \neq \rho_f$.

The following example shows that, if C is not distributive, then ω may not be join-uniform.

Example 2. Let M_3 be the 5-element lattice with 0, 1 and atoms a, b and c (see [12]). Define $f : M_3 \to M_3$ by $f(0) = f(a) = f(b) = 0$ and $f(c) = f(1) = 1$. Obviously f is monotone. One easily calculate that $\rho_f = 0$ and $\omega_f(0) = 0$ and $\omega_f(x) = 1$ for $x \in M_3 - \{0\}$. Since ω_f is additive, we have $\omega_f = \rho_{\omega_f}$. Thus $\rho_{\omega_f} = \omega_f \neq f = \rho_f$.

4 Systematic obscuring of DFA problems

Obfuscating a program P with respect to an analysis (or attacker) means to transform P into a functionally equivalent program P' such that the result of the analysis on P' is less precise than the result of the analysis on P. In fact, obfuscation with respect to an analysis can be elegantly formalized as a loss of precision of the analysis [14]. In the rest of this section we consider the standard data-flow analysis framework of Kildall and we instantiate it to the case of *reaching definition analysis*. Next we show that the reaching definition analysis is additive and, since the residuated ceiling of an additive function precisely corresponds to the maximal loss of precision with respect to additivity, we derive an obfuscating algorithm for the reaching definition analysis which is based on its residuated ceiling. This confirms the intuition that code obfuscation for an analysis is making the analysis maximally imprecise.

4.1 Kildall's monotone distributive framework for DFA

All the existing forward (backward) data-flow analyses, such as reaching definition, live variables, available expression, etc., consider the CFG of a program and are defined in terms of a pair of functions that specify the information that is true respectively at the entry and at the exit of each block (or program point) of the CFG. More specifically, these forward (resp. backward) data-flow analyses proceed as follows: (1) specify the information that holds at the start (resp. end) of a program; (2) if a node has more than one incoming (resp. outgoing) edge then combine the incoming (resp. outgoing)

information; (3) describe how the execution of a node changes the information which is propagated forward (reps. backward) from that node. Thus, a general framework for DFA consists of a domain D of data-flow facts that express the information of interest; an operator $\bigsqcup$ on the domain D for combining the information coming from multiple predecessors (successors); and a set $\mathcal{F}$ of (transfer) functions on D that describe how a node modifies the information flowing forward (res. backward) through that node. We consider here the general monotone and distributive framework for DFA introduced by Kildall [22] that put some additional requirements on the domain D and on the set of functions $\mathcal{F}$ in order to guarantee the correctness of the analysis.

Definition 1 ([22]). *A monotone distributive framework for DFA consists of:*

- *A complete lattice $(D, \leq)$ that satisfies the ascending chain condition, with least upper bound $\bigsqcup$;*
- *A set $\mathcal{F}$ of monotone functions form D to D that contains the identity function and that is closed under function composition. Moreover, each function f in $\mathcal{F}$ is distributive, which means that for every pair of elements d_1 and d_2 in D we have that $f(d_1 \sqcup d_2) = f(d_1) \sqcup f(d_2)$.*

Reaching Definition Analysis: Let us instantiate the general framework of Killdal at the case of reaching definition (RD) analysis. RD analysis is a forward DFA that for each node of the CFG is interested in characterizing the assignments that may have been made and not overwritten when reaching this node. We consider a program P as consisting of a sequence of instructions. Each node in the CFG of P represents an elementary block of P, namely a (maximal) sequence of instructions of P that are executed sequentially. We associate an unique location to every block of a program and write $[stmt_1; \ldots; stmt_n]^l$ to specify that the block at location l contains the sequence of statements $stmt_1; \ldots; stmt_n$. We consider the following set *Stmt* of possible program statements:

$$stmt ::= x := e \mid x := R \mid \texttt{case}\{(b_1, l_1), \ldots, (b_n, l_n)\} \mid \texttt{ret}\ x \mid \texttt{skip} \mid stmt; stmt$$

where $x := e$ assigns the value of expression e to variable x, $x := R$ assigns a random value from the set R to variable x; $\texttt{case}\{(b_1, l_1), \ldots, (b_n, l_n)\}$ implements a guarded multiple branch that redirects the flow of computation to the location l_i associated to the boolean condition b_i that evaluates to true. The other statements have the standard meaning. Let us denote with $Loc[P]$ the locations of the blocks of program P (where $|Loc[P]| \geq 2$); with $init[P]$ and $final[P]$ be the locations of the initial and final blocks of P; with $Block[P] \subseteq Stmt^* \times Loc[P]$ be the set of elementary blocks of program P, and with $Var[P]$ be the variables of P. Let $succ[P] : Stmt^* \times Loc[P] \rightarrow \wp(Loc[P])$ be a function that computes the locations of the possible successors of a block at a given location in a program P. Since the statements in a block are executed sequentially the successors of a block are determined by the forward flow of its last statement that is usually a `case` construct (except for the final block that has no successors). Let S denote a sequence of sequential statements.

- $succ[P]([S; \texttt{case}\{(b_1, l_1), \ldots, (b_n, l_n)\}]^l) = \{l_1, \ldots, l_n\}$;

– $succ[P]([S\,; stmt]^l) = \emptyset$ when $stmt$ is not a case construct.

The RD analysis is based on functions $kill$ and gen that compute the pairs of variables and labels that are *killed* and *generated* by the execution of each block:

– $kill([x := a]^l) = kill([x := R]^l) = \{(x, ?)\} \cup \{(x, l') \mid [S]^{l'}$ contains an assignment to $x\}$;
– $kill([S_1; S_2]^l) = kill([S_1]^l) \cup kill([S_2]^l)$;
– $kill([stmt]^l) = \emptyset$ in all the other cases;
– $gen([x := a]^l) = gen([x := R]^l) = \{(x, l)\}$;
– $gen([S_1; S_2]^l) = gen([S_1]^l) \cup gen([S_2]^l)$;
– $gen([stmt]^l) = \emptyset$ in all the other cases;

Since each block in a program is uniquely identified by its location, sometimes we write $kill(l)$ for $kill([S]^l)$, and $gen(l)$ for $gen([S]^l)$. The RD analysis of program P is defined by the pair of functions $RD_{entry}[P], RD_{exit}[P] : Loc[P] \to \wp(Var[P] \times Loc[P])$ that given a program location l, return respectively the locations that contain the definition of a variable that reaches the entry or the exit of the block at location l.

$$RD_{entry}[P](l) = \begin{cases} \{(x, ?) \mid x \in Var[P]\} & \text{if } l = init[P] \\ \bigcup \{RD_{exit}[P](l') \mid l \in succ[P](l')\} & \text{otherwise} \end{cases}$$

$$RD_{exit}[P](l) = (RD_{entry}[P](l) \setminus kill(l)) \cup gen(l)$$

By specifying function $RD_{exit}[P]$ with respect to a program location l we obtain function $RD_{exit}[P, l] : \wp(Var[P] \times Loc[P]) \to \wp(Var[P] \times Loc[P])$ that, by definition, behaves as follows:

$$RD_{exit}[P, l] = \lambda X.\ (X \setminus kill(l)) \cup gen(l).$$

Thus, when instantiating the general DFA framework to RD we have that the domain D of data-flow facts of interest is given by the complete lattice $\wp(Var[P] \times Loc[P])$, and the set $\mathcal{F}$ of monotone functions, that specify how the execution of a block modifies the analysis, is given by $\{RD_{exit}[P, l] \mid l \in Loc[P]\}$.

Example 3. Let us consider a program P that randomly assigns a value between 0 and 1 to variable b and then computes the sum of 9 and 5 if b is equal to 1, or the product of 9 and 2^5 if b is equal to 0. In Fig. 1 on the left we report the CFG of program P. The number at the top left of each block of the CFG of P denotes the location of the block. The RD analysis of P has the following solution:

– $RD_{entry}[P](1) = \{(x, ?), (y, ?), (b, ?), (t, ?)\}$
– $RD_{exit}[P](1) = \{(x, 1), (y, 1), (b, 1), (t, ?)\}$
– $RD_{entry}[P](2) = \{(x, 1), (x, 3), (y, 1), (y, 3), (b, 1), (t, ?)\} = RD_{exit}[P](2) = RD_{entry}[P](3)$
– $RD_{exit}[P](3) = \{(x, 3), (y, 3), (b, 1), (t, ?)\}$
– $RD_{entry}[P](4) = \{(x, 1), (x, 5), (y, 1), (y, 5), (b, 1), (t, ?)\}$
– $RD_{exit}[P](4) = \{(x, 1), (x, 5), (y, 1), (y, 5), (b, 1), (t, 4)\} = RD_{entry}[P](5)$
– $RD_{exit}[P](5) = \{(x, 5), (y, 5), (b, 1), (t, 4)\}$
– $RD_{entry}[P](6) = \{(x, 1), (x, 3), (x, 5), (y, 1), (y, 3), (y, 5), (b, 1), (t, ?), (t, 4)\} = RD_{exit}[P](6)$

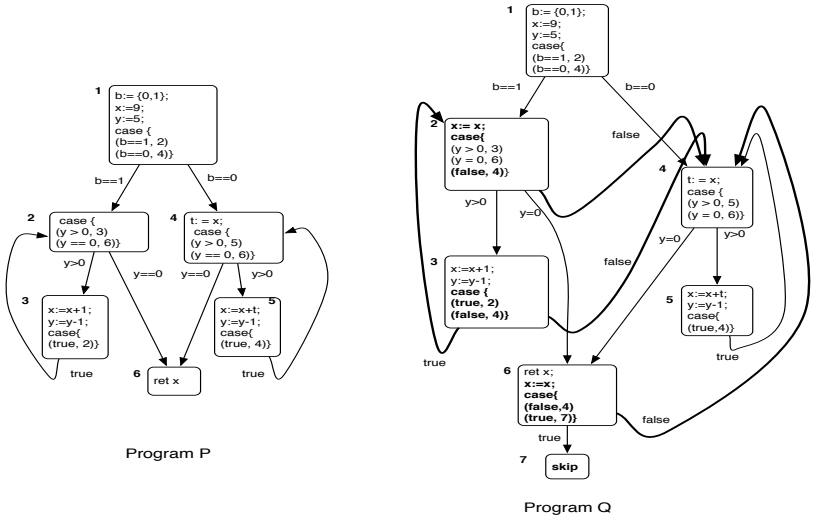

Fig. 1. The CFG of program P and of its obfuscated variant Q

4.2 Obfuscating RD Analysis

The RD analysis is clearly additive. More precisely, given a program location l, we have that function $RD_{exit}[P, l] : \wp(Var[P] \times Loc[P]) \to \wp(Var[P] \times Loc[P])$ is additive:

$$\begin{aligned} RD_{exit}[P, l](X_1 \cup X_2) &= ((X_1 \cup X_2) \setminus kill(l)) \cup gen(l) \\ &= (X_1 \setminus kill(l)) \cup gen(l)) \cup ((X_2 \setminus kill(l)) \cup gen(l)) \\ &= RD_{exit}[P, l](X_1) \cup RD_{exit}[P, l](X_2) \end{aligned}$$

This means that in order to make the RD analysis imprecise we have to become imprecise for disjunction when computing function $RD_{exit}[P, l]$. Thus, if we want to obfuscate a program P in order to make the analysis of RD imprecise at location l we have to transform P into a program Q such that the computation of function $RD_{exit}[Q, l]$ is maximally imprecise every time that, at the entry of location l, we have that a variable could have been defined in more than one location. This means that we have to design a program Q such that the computation of function $RD_{exit}[Q, l]$ corresponds to the maximal loss of additivity of function $RD_{exit}[P, l]$, namely we want a function $RD_{exit}[Q, l]$ to behave precisely like the residuated ceiling of $RD_{exit}[P, l]$. Observe that from Theorem 2 the existence of the residuated ceiling of function $RD_{exit}[P, l]$ is guaranteed by the fact that the domain $\wp(Var[P] \times Loc[P])$ is a dual algebraic and infinitely distributive complete lattice. For simplicity in the following we consider the RD analysis of a single variable x at a given location l of a program P, denoted $RD_{entry}[P, l, x]$ and $RD_{exit}[P, l, x]$. In particular: $RD_{exit}[P, l, x] = \lambda X.(X \setminus kill(x, l)) \cup gen(x, l)$ where $kill(l, x)$ and $gen(l, x)$ restrict the computation of functions $kill(l)$ and $gen(l)$ to the definition of the single variable x.

Following the definitions of the RD functions we have that if the block at location l of a given program P contains an assignment to variable x then $RD_{exit}[P, l, x] =$

$\lambda X.\{(x, l)\}$, which means that at the exit of the block the analysis is precise and returns the singleton $\{(x, l)\}$. Following this observation, we can obfuscate the RD analysis only for those variables that are not defined in the block that we want to obfuscate. Under this hypothesis, loosing precision in the computation of function $RD_{exit}[P, l, x]$ by loosing its additivity means that every time that the input X to the above function is not a singleton set then the analysis returns a "I do not know" value: *Any variable may reach that block*. Following this intuition, we define function $k[P, l, x] : \wp(Var[P] \times (Loc[P] \setminus \{l\})) \to \wp(Var[P] \times (Loc[P] \setminus \{l\}))$ as follows:

$$k[P, l, x] = \lambda X. \begin{cases} RD_{exit}[P, l, x](X) & \text{if } |X| \leq 1 \\ \top_l & \text{otherwise} \end{cases}$$

where $\top_l$ corresponds to the set $\{(x, i) \mid i \in Loc[P] \setminus \{l\}\}$, namely $\top_l$ means that variable x could have been defined in every block of the program but not in the one that we are currently analyzing. Observe that $k[P, l, x]$ is defined on $\wp(Var[P] \times (Loc[P] \setminus \{l\}))$ and this follows from the hypothesis that we are obfuscating the RD analysis for variable x that is not defined at location l. Function $k[P, l, x]$ is clearly non-additive:

$$\begin{aligned} k[P, l, x](\{(x, j)\}) \cup k[P, l, x](\{(x, i)\}) &= RD_{exit}[P, l, x](\{(x, j)\}) \cup RD_{exit}[P, l, x](\{(x, i)\}) \\ &\subset k[P, l, x](\{(x, j), (x, i)\}) = \top_l \end{aligned}$$

The next result shows that $k[P, l, x]$ is precisely the residuated ceiling of $RD_{exit}[P, l, x]$.

Theorem 3. *Consider a program P and assume that variable x is not defined in the block at location l of a given program P. Function $k[P, l, x]$ is the residuated ceiling of $RD_{exit}[P, l, x]$, i.e., $k[P, l, x] = \omega(RD_{exit}[P, l, x])$. Namely $k[P, l, x]$ is the most abstract function such that $\rho(k[P, l, x]) = \rho(RD_{exit}[P, l, x]) = RD_{exit}[P, l, x]$.*

This means that if we want to obfuscate the RD analysis of a program P at a given location l for a given variable x, we have to modify the program in such a way that the RD analysis at l of x of the transformed program Q behaves like the residuated ceiling $\omega(RD_{exit}[P, l, x]) = k[P, l, x]$. This allows us to state the following result that proves optimality and provides a mathematical foundation of the intuitive obfuscation of RD.

Theorem 4. *Let P be a source program. A transformed program Q (with the same functionality of P) maximally obfuscates the RD analysis with respect to variable x at location l of program P if $RD_{exit}[Q, l, x] = \omega(RD_{exit}[P, l, x])$.*

Example 4. Let us design a program Q that obfuscates the RD analysis of the variable x at the block 4 of the program P presented in Example 3, namely we want that $\omega(RD_{exit}[P, 4, x]) = RD_{exit}[Q, 4, x]$. Since $RD_{entry}[P, 4, x] = \{(x, 1), (x, 5)\}$ then we want that $RD_{exit}[Q, 4, x] = \top_4$. In order to obtain this we have to design the program Q in such a way that every block (except for the one at location 4) contains a definition of x and that this definition reaches the block at location 4 of program Q. In Fig. 1 on the right we show a program Q that satisfies these requirements. In fact, the RD analysis of program Q with respect to variable x at program point 4 returns $RD_{exit}[Q, x, 4] = \{(x, 1), (x, 2), (x, 3), (x, 5), (x, 6)\} = \top_4$. Observe that the two programs have the same functionality since the modifications made to P to obtain Q preserve program functionality.

Following the previous example we can derive an algorithm that maximally obfuscates the RD analysis of a program in a given location with respect to a particular variable. Given a block $[S]^l$, which is not final, we denote with $[S \oplus \texttt{case}\{(b, l')\}]^l$ the block that we obtain from $[S]^l$ by adding the pair (b, l') to the `case` construct at the end of the sequence of statements inside the elementary block.

Optimal RD-obfusaction

1 : **input**: $Block[P]$, l, x;
2 : $A := Block[P] \setminus \{ [S]^l \} \cup \{ [S]^j \mid (x, j) \in RD_{exit}[P, l, x] \}$;
3 : **while** $A \neq \emptyset$ **do**
4 : select a block $[S]^i$ from A;
5 : **if** $i \in final[P]$
6 : **then** $Block[P] := Block[P] \setminus \{[S]^i\} \cup \{[x := x; S; \texttt{case}\{(true, l_{new}), (false, l)\}]^i, [skip]^{l_{new}}\}$
7 : **else** $Block[P] := Block[P] \setminus \{[S]^i\} \cup \{[x := x; S \oplus \texttt{case}\{(false, l)\}]^i\}$;
8 : $A := A \setminus [S]^i$;
9 : **output**: $Block[P]$

The algorithm **Optimal RD-obfusaction** takes as inputs a program P in the form of its elementary blocks, a program location l and a variable x (not defined in the block l) and returns the blocks of a program with the same functionality of P that maximally obfuscates the RD analysis of variable x at location l. In order to do this, the algorithm defines at line 2 the set A of the elementary blocks of P that are not at the location that we want to obfuscate and that do not already contain a definition of variable x that reaches location l. The algorithm modifies each block in A in order to ensure that it contains a definition of variable x that reaches location l. We distinguish two cases. At line 7 of the algorithm we consider the case when the block $[S]^i$ is not final and we add (at the beginning of the block) the assignment $x := x$ (which is clearly a semantic nop), and we enrich the `case` construct that terminates the block with the pair $(false, l)$ which makes the block at location l reachable from the current block (even if at execution time this never happens thus preserving the semantics of the program). At line 6 of the algorithm we consider the case when the block $[S]^i$ is final (namely it has no successors). In this case the block $[S]^i$ does not end with a `case` construct and therefore we add (at the beginning of the block) the assignment $x := x$ and at the end of the block a case construct $\texttt{case}\{(true, l_{new}), (false, l)\}$, where l_{new} is a new program location. By doing this the block at location i is no longer final and therefore we add a final block at the new location l_{new} that contains only the *skip* statement.

The above algorithm for the obfuscation of RD analysis is very simple and easy to break. In order to make it more resilient we could use sophisticated opaque predicates instead of the always false condition *false*, and more complex obfuscations for the insertion of the semantic nop instead of the simple $x := x$. In the literature we can find many obfuscation techniques for opaque predicates insertion and semantic nops insertion, such as in [5, 23, 29, 30].

5 Conclusion

We introduced basic function transformers that respectively induce and remove additivity from functions. These operations have been proved to play a key role in modeling an important aspect in static program analysis, which is additivity. In particular we proved that the residuated ceiling provides a mathematical foundation of standard code obfuscation strategies defeating distributive data-flow analysis problems, such as reaching definitions (RD) analysis. This confirms the intuition in [14]: *Making a program obscure for an analysis is making the same analysis imprecise for the transformed code*.

Residuated ceilings express here the imprecision in disjunction, which is distributivity in monotone frameworks. Other algorithms exist for breaking distributivity. Wang et al. [30] present a code obfuscation technique based on control flow flattening and variable aliasing that drastically reduces the precision of static analysis. Their basic idea is to make the analysis of the program control flow dependent on the analysis of the program data-flow, and then to use aliasing to complicate data-flow analysis. In particular, the proposed obfuscation transforms the original control flow of the program into a flattened one where each elementary block can be the successor/predecessor of any other elementary block in the CFG of the program. In order to preserve the semantics of the program, the actual program flow is determined dynamically by a dispatcher. This obfuscating transformation clearly also obfuscates the RD analysis. However, it is not the simplest obfuscation for disjointness in RD. Theorem 4 proves that this can be achieved by the residuated ceiling of RD analysis. A far more elementary analysis (an therefore a wider number of attacks) is obscured in Wang's transformation, as recently proved in [15], which is precisely the Control-Flow Graph extraction analysis, which is used in RD analysis. This of course obscures RD in the sense of lack of precision. In fact, it makes every elementary block of the CFG reachable from every other one, so the RD analysis of a flattened program would conclude that the definition of a variable reaches all the elementary blocks of the program. This means that for every location l we have that $RD_{entry}[P^f](l) = \top_l$, and this means that for every variable x that is not defined in the block at location l we have that $RD_{exit}[P^f, l, x] = \omega(RD_{exit}[P, l, x])$, where P is the original program (before the control flow flattening obfuscation).

References

1. H. Andréka, R. J. Greechie, and G. E. Strecker, "On residuated approximations", in *Categorical Methods in Computer Science With Aspects from Topology,* 1989, pp 333-339.
2. V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations", in *PLDI* pp 41-53, New York, NY, 1989.
3. G. Birkhoff, "Subdirect unions in universal algebra", Bull. Amer. Math. Soc. 50 (1944), pp 764-768.
4. T. Blyth and M. Janowitz, "Residuation theory", Pergamon Press, 1972.
5. C. Collberg and C. Thomborson and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs, In *25th POPL*, pp 184 -196, ACM Press, 1998.
6. P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points", in *4th POPL*, 1977, pp 238-252.

7. P. Cousot and R. Cousot, “Abstract Interpretation framework”, *Journal of Logic and Computation*, 2(4): 511-547 August 1992.
8. P. Cousot and R. Cousot, “Static determination of dynamic properties of programs”, in *Proc. 2nd Int. Symp. on Programming*, pp 106-130, Dunod, 1976.
9. P. Cousot and R. Cousot, “Systematic design of program analysis frameworks”, in 6th *POPL*, pp 269-282, San Antonio, TX, 1979, ACM Press.
10. P. Cousot and N. Halbwachs, “Automatic discovery of linear restraints among variables of a program”, in 5th *POPL*, pp 84-97, Tucson, AZ, 1978, ACM Press.
11. M. Dalla Preda and R. Giacobazzi, “Semantics-based code obfuscation using abstract interpretation”, *Journal of Computer Security*, 17(6): 855-908, 2009.
12. B. A. Davey and H. A. Priestly, “Introduction to Lattices and Order”, Cambridge University Press, 2002.
13. W. Feng, J. Su and R. Greechie, “On calculating residuated approximations”, Algebra Universalis, to appear.
14. R. Giacobazzi. “Hiding information in completeness holes - New perspectives in code obfuscation and watermarking”, in 6th *IEEE International Conference SEFM*, pp 7-20, 2008.
15. R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by Partial Evaluation of Distorted Interpreters. In Proc. *ACM SIGPLAN Partial Evaluation and Program Manipulation (PEPM’12)*, Philadelphia USA, 2012. To appear.
16. R. Giacobazzi and F. Ranzato, “Uniform Closures: Order-Theoretically Reconstructing Logic Program Semantics and Abstract Domain Refinements”, *Information and Computation*, Vol 145, No. 2, pp 153-190, 1998.
17. R. Giacobazzi and F. Ranzato, “Optimal domains for disjunctive abstract interpretation”, *Science of Computer Programming*, Vol 32, No. 1-3, pp 177-210, 1998.
18. G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, 1980.
19. P. Granger, “Static analysis of arithmetic congruence”, *Int. J. Comput. Math.*, 30: 165-190, 1989.
20. P. Granger, “Static analysis of linear congruence equalities among variables of a program”, in S. Abramsky and T. S. E Maibaum, editors, *Proc. Int. J. Conf. TAPSOFT’91, Volume 1 (CAAP’91)*, Brighton, GB, LNCS 493, pp 169-192, Springer-Verlag, 1991.
21. R. J. Greechie and M. F. Janowitz, personal communication, 2005.
22. G. A. Kildall, “Global expression optimization during compilation”, in 1st *POPL*, pp 194-206, Boston, MA, 1973, ACM Press.
23. A. Majumdar and C. Thomborson. ”Manufactoring opaque predicates in distributed systems for code obfuscation”, In 29th Australasian Computer Science Conference (ACSC’06), volume 48, pages 187-196, 2006.
24. F. Masdupuy, “Semantic analysis of interval congruences”, in *Proc FMPA*, Akademgorodok, Novosibirsk, RU, LNCS 735, pp 142-155. Springer-Verlag, 1993.
25. L. Mauborgne, “Tree schemata and fair termination”, in *Proc.* 7th *Int. Symp. SAS’2000*, Santa Barbara, CA, US, LNCS 1824, pp 302-321. Springer-Verlag, 29, 2000.
26. F. Nielson, H. Nielson and C. Hankin, “Principles of Program Analysis”, Springer, 2004.
27. D. A. Plaisted “Theorem proving with abstraction”, *Artificial Intelligence*, Volume 16, pp 47-108, March 1981.
28. J. Su, W. Feng and R. J. Greechie, ”Disitributivity conditions and the order-skeleton of a lattice”, Algebra Universalis, Vol 66, No 4: 337-354, 2011.
29. P. Szor, ”The Art of Computer Virus Research and Defense”, Addison-Wesley Professional, 2005.
30. C. Wang and J. Hill and J. Knight and J. Davidson. ”Software tamper resistance: obstructing static analysis of programs”, Technical Report CS-2000-12, Department of Computer Science, University of Virginia, 2000.

A Technical proofs

Let $A_f(x) := \{q \in Q \mid x \leq \bigvee f^{-1}(\downarrow q)\}$ so that $\sigma_f(x) = \bigwedge A_f(x)$. Recall that $\downarrow S = \bigcup_{s \in S} \downarrow s$.

Lemma 1. Let C be an infinitely distributive complete lattice and let $S \subseteq C$. If $x \leq \bigvee S$ and $x \in JI(C)$, then $x \in \downarrow S$. Hence $(\downarrow S) \cap JI(C) = (\downarrow \bigvee S) \cap JI(C)$ holds.

Proof. Let $S \subseteq C$ with $x \leq \bigvee S$. Since C is infinitely distributive, we have $x = x \wedge (\bigvee S) = \bigvee_{s \in S} (x \wedge s)$; since $x \in JI(C)$, it follows that $x \in \{x \wedge s \mid s \in S\}$, so that $x = x \wedge s$, i.e., $x \leq s$, for some $s \in S$, proving the first statement. The second statement follows immediately.

Lemma 2. Let $f : C \to A$ be a monotone mapping from an infinitely distributive complete lattice C to a complete lattice A. Then, for all $x \in JI(C)$, $f(x) = \sigma_f(x) = \rho_f(x)$.

Proof. Assume that $x \in JI(C)$. Let $q \in A_f(x)$, then $x \leq \bigvee f^{-1}(\downarrow q)$; by Lemma 1, there is some $s \in f^{-1}(\downarrow q)$ such that $x \leq s$, so that $f(x) \leq f(s) \leq q$, since f is monotone. Hence $f(x) \leq \bigwedge A_f(x) = \sigma_f(x)$. Since also $\sigma_f(x) \leq f(x)$, we have $\sigma_f(x) = f(x)$. By induction we have $\sigma_f^{(\alpha)}(x) = f(x)$ for $\alpha \geq 1$. Therefore $\rho_f(x) = \sigma_f(x) = f(x)$.

Theorem 1. Let $f : C \to A$ be a monotone mapping from a dual-algebraic infinitely distributive complete lattice C to a complete lattice A. Then, for all $x \in C$,

$$\rho_f(x) = \sigma_f(x) = \bigvee f((\downarrow x) \cap JI(C))$$

Proof. Let $x \in C$ and let $M(x) := (\downarrow x) \cap JI(C)$. Since C is dual algebraic, we have $x = \bigvee M(x)$. Observe that, for all $S \subseteq C$, $f(S) \subseteq \downarrow \bigvee f(S)$, i.e. $S \subseteq f^{-1}(\downarrow \bigvee f(S))$. Then $M(x) \subseteq f^{-1}(\downarrow \bigvee f(M(x)))$, so that $x = \bigvee M(x) \leq \bigvee f^{-1}(\downarrow \bigvee f(M(x)))$. It follows that $\bigvee f(M(x)) \in A_f(x)$, thus $\sigma_f(x) = \bigwedge A_f(x) \leq \bigvee f(M(x))$.

Let $q \in A_f(x)$. Then $x \leq \bigvee f^{-1}(\downarrow q)$ and $(\downarrow x) \cap JI(C) \subseteq (\downarrow \bigvee f^{-1}(\downarrow q)) \cap JI(C)$; since $(\downarrow \bigvee f^{-1}(\downarrow q)) \cap JI(C) = (\downarrow f^{-1}(\downarrow q)) \cap JI(C)$ by Lemma 1, we have $(\downarrow x) \cap JI(C) \subseteq (\downarrow f^{-1}(\downarrow q)) \cap JI(C)$ and $\bigvee M(x) \leq \bigvee M(f^{-1}(\downarrow q))$. Since, for $y \in M(x)$, $y \leq \bigvee M(x) \leq \bigvee M(f^{-1}(\downarrow q)) = \bigvee((\downarrow f^{-1}(\downarrow q)) \cap JI(C))$, there exists $s \in f^{-1}(\downarrow q)$ such that $y \leq s$, by Lemma 1 part (1); thus $f(y) \leq f(s) \leq q$. It follows that $\bigvee f(M(x)) \leq q$, so that $\bigvee f(M(x)) \leq \bigwedge A_f(x) = \sigma_f(x)$. Therefore $\sigma_f(x) = \bigvee f(M(x)) = \bigvee f((\downarrow x) \cap JI(C))$.

By replacing f with σ_f, it follows that $\sigma_f^{(2)}(x) = \sigma_{\sigma_f}(x) = \bigvee \sigma_f((\downarrow x) \cap JI(C))$; we have $\bigvee \sigma_f((\downarrow x) \cap JI(C)) = \bigvee f((\downarrow x) \cap JI(C))$ from Lemma 2. Hence $\sigma_f^{(2)}(x) = \bigvee f((\downarrow x) \cap JI(C)) = \sigma_f(x)$. By induction we have $\sigma_f^{(\alpha)}(x) = \sigma_f(x)$ for all $\alpha \geq 1$. Therefore $\rho_f(x) = \sigma_f(x)$.

Theorem 2. Let C be a dual-algebraic complete lattice and A be a complete lattice. If C is infinitely distributive, then ρ is join-uniform.

Proof. Let $f \in \mathcal{M}_{C,A}$. It is easy to see that $\omega_f \in \mathcal{M}_{C,A}$. We claim that $\rho_{\omega_f} = \rho_f$. Let $x \in C$ and let $M(x) := (\downarrow x) \cap JI(C)$. Then $\rho_{\omega_f}(x) = \bigvee \omega_f(M(x))$ and $\rho_f(x) = \bigvee f(M(x))$ by Theorem 1. Let $y \in M(x)$. Since $g(y) = \rho_g(y)$ for all $g \in \mathcal{M}_{C,A}$ by Lemma 2, we have $\omega_f(y) = \bigvee\{\rho_g(y) \mid g \in \mathcal{M}_{C,A} \text{ and } \rho_g = \rho_f\} = \rho_f(y)$, so that $\rho_{\omega_f}(x) = \bigvee \omega_f(M(x)) = \bigvee \rho_f(M(x))$. Since $f(y) = \rho_f(y)$ for all $y \in M(x)$ by Lemma 2, it follows that $\rho_{\omega_f}(x) = \bigvee f(M(x)) = \rho_f(x)$. Therefore ρ is join-uniform.

Theorem 3. Consider a program P and assume that variable x is not defined in the block at location l of the given program P. Function $k[P,l,x]$ is the residuated ceiling of $RD_{exit}[P,l,x]$. Namely $k[P,l,x]$ is the most abstract function such that $\rho(k[P,l,x]) = \rho(RD_{exit}[P,l,x]) = RD_{exit}[P,l,x]$.

Proof. Under the hypothesis that variable x is not defined in the block at location l we have that function $k[P,l,x] : \wp(Var[P] \times (Loc[P] \setminus \{l\})) \to \wp(Var[P] \times (Loc[P] \setminus \{l\}))$ is monotone. Observe that the fact that variable x is not defined in the block at location l implies that $kill(x,l) = gen(x,l) = \emptyset$ and therefore that $RD_{exit}[P,l,x](X) = X$. This means that function $k[P,l,x]$ behaves like the identity on every element X such that $|X| \leq 1$. Given two elements $X_1, X_2 \in \wp(Var[P] \times (Loc[P] \setminus \{l\}))$, we distinguish the following cases:

- $X_1 \subseteq X_2$ and $|X_1| > 1$ and $|X_2| > 1$: in this case monotonicity is guaranteed by the fact that $k[P,l,x](X_1) = k[P,l,x](X_2) = \top_l$;
- $X_1 \subseteq X_2$ and $|X_1| \leq 1$ and $|X_2| \leq 1$: this implies that $k[P,l,x](X_1) = X_1$ and that $k[P,l,x](X_2) = X_2$ from which it follows monotonicity;
- $X_1 \subseteq X_2$ and $|X_1| \leq 1$ and $|X_2| > 1$: in this case we have that $k[P,l,x](X_1) = X_1$ and $k[P,l,x](X_2) = \top_l = \{(x,i) \mid i \neq l\}$. We have two possible cases: (A) If $|X_1| = 0$ then monotonicity follows form the fact that $k[P,l,x](X_1) = X_1 = \emptyset \subseteq \top_l$; (B) If $|X_1| = 1$ then we have $X_2 = \{(x,j)\}$ where $j \in Loc[P]$ and $j \neq l$, and therefore monotonicity follows form the fact that $k[P,l,x](X_1) = X_1 = \{(x,j)\} \subseteq \top_l$.

Let us recall that given a function $f : A \to B$ we say that for $S \subseteq A$ then $f(S) = \cup_{s \in S} f(s)$. Function $RD_{exit}[P,l,x]$ is additive and therefore $\rho(RD_{exit}[P,l,x]) = RD_{exit}[P,l,x]$. Let us compute $\rho(k[P,l,x])$. The domain $\wp(Var[P] \times Loc[P])$ is a dual algebraic infinitely distributive complete lattice, function $RD_{esxit}[P,l,x]$ is monotone, therefore we can apply Theorem 1 and we conclude that:

$$\begin{aligned}
\rho(k[P,l,x])(X) &= \bigcup k[P,l,x]((\downarrow X) \cap JI(\wp(Var[P] \times Loc[P]))) \\
&= \bigcup k[P,l,x](\{\, \{(x,i)\} \mid (x,i) \in X \,\}) \\
&= \bigcup \left\{ k[P,l,x](\{(x,i)\}) \,\middle|\, (x,i) \in X \right\} \\
&= \bigcup \left\{ RD_{exit}[P,l,x](\{(x,i)\}) \,\middle|\, (x,i) \in X \right\} \\
&= RD_{exit}[P,l,x](X)
\end{aligned}$$

In order to conclude the proof we have to show that function $k[P,l,x]$ is the most abstract function whose residuated approximation is $RD_{exit}[P,l,x]$. Let us consider a function $h[P,l,x] : \wp(Var[P] \times Loc[P]) \to \wp(Var[P] \times Loc[P])$ such that $k[P,l,x] \sqsubset h[P,l,x]$. Since $k[P,l,x]$ returns $\top_l$ on every input that is not a singleton, the fact that $h[P,l,x]$ is bigger than $k[P,l,x]$ means that there exists a singleton $\{(x,i)\} \in \wp(Var[P] \times Loc[P])$ such that $k[P,l,x](\{(x,i)\}) = RD_{exit}[P,l,x](\{(x,i)\}) \subset h[P,l,x](\{(x,i)\})$. Therefore, when computing the residuated approximation of $h[P,l,x]$ on the element $\{(x,i)\} \in \wp(Var[P] \times Loc[P])$ we have that $\rho(h[P,l,x])(\{(x,i)\}) = h[P,l,x](\{(x,j)\})$ and, by definition of $h[P,l,x]$ we have $RD_{exit}[P,l,x](\{(x,i)\}) \subset h[P,l,x](\{(x,i)\})$. This means that $RD_{exit}[P,l,x] \sqsubset \rho(h[P,l,x])$, which concludes the proof.